Practical Software Diversification for WebAssembly

JAVIER CABRERA-ARTEAGA

Doctoral Thesis in Computer Science
Supervised by
Benoit Baudry and Martin Monperrus
Stockholm, Sweden, 2023

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
Division of Software and Computer Systems
SE-10044 Stockholm
Sweden

TRITA-EECS-AVL-2020:4
ISBN 100-

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av Teknologie doktorexamen i elektroteknik i .

Tryck: Universitetsservice US AB

Abstract

Keywords: Lorem, Ipsum, Dolor, Sit, Amet

Sammanfattning

LIST OF PAPERS

1. ***WebAssembly Diversification for Malware Evasion***
Javier Cabrera-Arteaga,Tim Toady, Martin Monperrus, Benoit Baudry
Computers & Security, Volume 131, 2023, 17 pages
`https://www.sciencedirect.com/science/article/pii/S0167404823002067`

2. ***Wasm-mutate: Fast and Effective Binary Diversification for WebAssembly***
Javier Cabrera-Arteaga, Nicholas Fitzgerald, Martin Monperrus, Benoit Baudry
Under review, 17 pages
`https://arxiv.org/pdf/2309.07638.pdf`

3. ***Multi-Variant Execution at the Edge***
Javier Cabrera-Arteaga,Pierre Laperdrix, Martin Monperrus, Benoit Baudry
Moving Target Defense (MTD 2022), 12 pages
`https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

4. ***CROW: Code Diversification for WebAssembly***
Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
Measurements, Attacks, and Defenses for the Web (MADWeb 2021), 12 pages
`https://doi.org/10.14722/madweb.2021.23004`

5. ***Superoptimization of WebAssembly Bytecode***
Javier Cabrera-Arteaga,Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs, 4 pages
`https://doi.org/10.1145/3397537.3397567`

6. ***Scalable Comparison of JavaScript V8 Bytecode Traces***
Javier Cabrera-Arteaga,Martin Monperrus, Benoit Baudry
11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019), 10 pages
`https://doi.org/10.1145/3358504.3361228`

ACKNOWLEDGEMENT

Contents

Part I

Thesis

1 INTRODUCTION

TODO Recent papers first. Mention Workshops instead in conference. "Procedings of XXXX". Add the pages in the papers list.

1.1 Background

TODO Motivate with the open challenges.

1.2 Problem statement

TODO Problem statement TODO Set the requirements as R1, R2, then map each contribution to them.

1.3 Automatic Software diversification requirements

1. **1:** TODO Requirement 1

1.4 List of contributions

C1: Methodology contribution: We propose a methodology for generating software diversification for WebAssembly and the assessment of the generated diversity.

C2: Theoretical contribution: We propose theoretical foundation in order to improve Software Diversification for WebAssembly.

C3: Automatic diversity generation for WebAssembly: We generate WebAssembly program variants.

[0] Comp. time 2023/10/06 12:22:33

Contribution	Resarch papers				
	P1	P2	P3	P4	P5
C1	x	x		x	x
C2	x	x			
C3	x	x	x		
C4	x	x	x		
C5			x		
C6	x	x	x	x	x

Table 1.1: *Mapping of the contributions to the research papers appended to this thesis.*

C4: Software Diversity for Defensive Purposes: We assess how generated WebAssembly program variants could be used for defensive purposes.

C5: Software Diversity for Offensives Purposes: We assess how generated WebAssembly program variants could be used for offensive purposes, yet improving security systems.

C6: Software Artifacts: We provide software artifacts for the research community to reproduce our results.

TODO Make multi column table

1.5 Summary of research papers

P1: Superoptimization of WebAssembly Bytecode.

P2: CROW: Code randomization for WebAssembly bytecode.

P3: Multivariant execution at the Edge.

P4: Wasm-mutate: Fast and efficient software diversification for WebAssembly.

P5: WebAssembly Diversification for Malware evasion.

1.6 Thesis outline

2 BACKGROUND AND STATE OF THE ART

THIS chapter discusses the state-of-the-art in the areas of WebAssembly and Software Diversification. In Section 2.1 ... In Section 2.2 ... TODO Describe chapter

TODO Add some words on the evergreen method of Wasm

2.1 WebAssembly

The W3C publicly announced the WebAssembly (Wasm) language in 2017 as the four scripting language supported in all major web browser vendors. Wasm is a binary instruction format for a stack-based virtual machine and was officially consolidated by the work of Haas et al. [?] in 2017 and extended by Rossberg et al. in 2018 [?]. It is designed to be fast, portable, self-contained and secure, and it promises to outperform JavaScript execution. Since 2017, the adoption of Wasm keeps growing. For example; Adobe, announced a full online version of Photoshop[1] written in WebAssembly; game companies moved their development from JavaScript to Wasm like is the case of a full Minecraft version[2].

Moreover, WebAssembly has been evolving outside web browsers since its first announcement. Some works demonstrated that using WebAssembly as an intermediate layer is better in terms of startup and memory usage than

[0] Comp. time 2023/10/06 12:22:33

[1] https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecA0K4V8R69lw

[2] https://satoshinm.github.io/NetCraft/

containerization and virtualization [? ?]. Consequently, in 2019, the Bytecodealliance proposed WebAssembly System Interface (WASI) [?]. WASI pioneered the execution of Wasm with a POSIX system interface protocol, making it possible to execute Wasm closer to the underlying operating system. Therefore, it standardizes the adoption of Wasm in heterogeneous platforms [?], making it suitable for standalone and backend execution scenarios [? ?].

2.1.1 From source code to WebAssembly

WebAssembly programs are compiled from source languages like C/C++, Rust, or Go, which means that it can benefit from the optimizations of the source language compiler. The resulting Wasm program is like a traditional shared library, containing instruction codes, symbols, and exported functions. A host environment is in charge of complementing the Wasm program, such as providing external functions required for execution within the host engine. For instance, functions for interacting with an HTML page's DOM are imported into the Wasm binary when invoked from JavaScript code in the browser.

In Listing 2.1 and Listing 2.2, we illustrate a C program and its corresponding Wasm binary. The C function includes heap allocation, external function usage, and a function definition featuring a loop, conditional branching, function calls, and memory accesses. The Wasm code in Listing 2.2 displays the textual format of the generated Wasm (Wat)[3].

```c
// Some raw data
const int A[250];

// Imported function
int ftoi(float a);

int main() {
    for(int i = 0; i < 250; i++) {
        if (A[i] > 100)
            return A[i] + ftoi(12.54);
    }

    return A[0];
}
```

Listing 2.1: *Example C program which includes heap allocation, external function usage, and a function definition featuring a loop, conditional branching, function calls, and memory accesses.*

[3] The WAT text format is mostly for human readability and for low-level manual modification.

```
; WebAssembly magic bytes(\0asm) and version (1.0) ;
(module
; Type section: 0x01 0x00 0x00 0x00 0x13 ... ;
  (type (;0;) (func (param f32) (result i32)))
  (type (;1;) (func))
  (type (;2;) (func (result i32)))
; Import section: 0x02  0x00 0x00 0x00 0x57 ... ;
  (import "env" "ftoi" (func $ftoi (type 0)))
; Custom section: 0x00 0x00 0x00 0x00 0x7E ;
  (@custom "name" "...")
; Code section: 0x03 0x00 0x00 0x00 0x5B... ;
  (func $main (type 2) (result i32)
    (local i32 i32)
    i32.const -1000
    local.set 0
    block  ;label = @1;
      loop  ;label = @2;
        i32.const 0
        local.get 0
        i32.add
        i32.load
        local.tee 1
        i32.const 101
        i32.ge_s
        br_if 1 ;@1;
        local.get 0
        i32.const 4
        i32.add
        local.tee 0
        br_if 0 ;@2;
      end
      i32.const 0
      return
    end
    f32.const 0x1.9147aep+3
    call $ftoi
    local.get 1
    i32.add)
; Memory section: 0x05 0x00 0x00 0x00 0x03 ... ;
  (memory (;0;) 1)
; Global section: 0x06 0x00 0x00 0x00 0x11.. ;
  (global (;4;) i32 (i32.const 1000))
; Export section: 0x07 0x00 0x00 0x00 0x72 ... ;
  (export "memory" (memory 0))
  (export "A" (global 2))
; Data section: 0x0d 0x00 0x00 0x03 0xEF  ... ;
  (data $data (0) "\00\00\00\00...")
; Custom section: 0x00 0x00 0x00 0x00 0x2F ;
  (@custom "producers" "...")
)
```

Listing 2.2: *Wasm code for Listing 2.1. The example Wasm code illustrates the translation from C to Wasm in which several high-level language features are translated into multiple Wasm instructions.*

2.1.2 WebAssembly’s binary format

The Wasm binary format is close to machine code and already optimized, being a consecutive collection of sections. In Figure 2.1 we show the binary format of a Wasm section. A Wasm section starts with a 1-byte section ID, followed by a 4-byte section size, and concludes with the section content, which precisely matches the size indicated earlier. A Wasm binary contains sections of 13 types, each with a specific semantic role and placement within the module. Each section is optional, where an omitted section is considered empty. In the following text, we summarize each one of the 13 types of Wasm sections, providing their name, ID, and purpose. In addition, some sections are annotated as comments in the Wasm code in Listing 2.2.

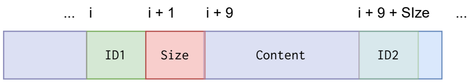

Figure 2.1: *Memory byte representation of a WebAssembly binary section, starting with a 1-byte section ID, followed by an 8-byte section size, and finally the section content.*

Custom Section (00) : Comprises two parts: the section name and arbitrary content. Primarily used for storing metadata, such as the compiler used to generate the binary (see lines 9 and 48 of Listing 2.2). This type of section has no order constraints with other sections and is optional. Compilers usually skip this section when consuming a WebAssembly binary.

Type Section (01) : Contains the function signatures for functions declared or defined within the binary (see lines 3 to 6 in Listing 2.2). Functions may share the same function signature. This section must occur only once in a binary. It can be empty.

Import Section (02) : Lists elements imported from the host, including functions, memories, globals, and tables (see line 8 in Listing 2.2). This section is needed to enable code and data sharing with the host engine and other modules. It must occur only once in a binary. It can be empty.

Function Section (03) : Details functions defined within the binary. It essentially maps Type section entries to Code section entries. The text format already maps the function index to its name, as shown in lines 12 to 38 of Listing 2.2. This section must occur only once in a binary and, it can be empty.

Table Section (04) : Groups functions with identical signatures to control indirect calls. It must occur only once in a binary. It can be empty. The example code in Listing 2.2 does not include a Table Section.

Memory Section (05) : Specifies the number and initial size of unmanaged linear memories (see line 40 in Listing 2.2). It must occur only once in a binary. It can be empty.

Global Section (06) : Defines global variables as managed memory for use and sharing between functions in the WebAssembly binary (see line 42 of Listing 2.2). It must occur only once in a binary. It can be empty.

Export Section (`07`) : Declares elements like functions, globals, memories, and tables for host engine access (see lines 44 and 45 of Listing 2.2). It must occur only once in a binary. It can be empty.

Start Section (`08`) : Designates a function to be called upon binary readiness, initializing the WebAssembly program state before executing any exported functions. It must occur only once in a binary. It can be empty. The example code in Listing 2.2 does not include a Start Section, i.e. there is no function to call when the binary is initialized.

Element Section (`09`) : Contains elements to initialize the binary tables. It must occur only once in a binary. It can be empty. The example code in Listing 2.2 does not include an Element Section.

Code Section (`10`) : Contains the body of functions defined in the Function section. Each entry consists of local variables used and a list of instructions (see lines 12 to 38 in Listing 2.2). It must occur only once in a binary. It can be empty.

Data Section (`11`) : Holds data for initializing unmanaged linear memory. Each entry specifies the offset and data to be placed in memory (see line 47 in Listing 2.2). It must occur only once in a binary. It can be empty.

Data Count Section (`12`) : Primarily used for validating the Data Section. If the segment count in the Data Section mismatches the Data Count, the binary is considered malformed. The example code in Listing 2.2 does not include a Data Count Section. It must occur only once in a binary. It can be empty.

Due to its organization into a contiguous array of sections, a Wasm binary can be processed efficiently. For example, this structure allows compilers to speed up the compilation process through parallel parsing or just by ignoring *Custom Sections*. Additionally, the use of the LEB128[4] encoding of instructions of the *Code Section* further compacts the binary. As a result, Wasm binaries are not only fast to validate and compile but also quick to transmit over a network.

2.1.3 WebAssembly's runtime structure

The WebAssembly runtime structure is described in the WebAssembly specification by enunciating 10 key components: the Store, Module Instances, Table Instances, Export Instances, Import Instances, the Execution Stack, Memory Instances, Global Instances, Function Instances and Locals. These

[4] `https://en.wikipedia.org/wiki/LEB128`

components are particularly significant in maintaining the state of a WebAssembly program during its execution. In the following text, we provide a brief description of each runtime component. Notice that, the runtime structure is an abstraction that serves to validate the execution of a Wasm binary.

Store : The WebAssembly store represents the global state and is a collection of instances of functions, tables, memories, and globals. Each of these instances is uniquely identified by an address, which is usually represented as an i32 integer.

Module Instances : A module instance is a runtime representation of a loaded and initialized WebAssembly module (the binary file described in Subsection 2.1.2). It contains the runtime representation of all the definitions within a module, including functions, tables, memories, and globals, as well as the module's exports and imports.

Table instances : A table instance is a vector of *function instances* with the same signature. They are used to validate and support indirect function calls during runtime. A table instance can be modified through table instructions from the function bodies.

Export Instances : Export instances represent the functions, tables, elements, globals or memories that are exported by a Wasm binary to the host environment.

Import Instances : Import instances represent the functions, tables, elements, globals or memories that are imported into a module from the host environment.

The Execution Stack holds typed values, labels and control frames, with labels handling block instructions, loops, and function calls. Values inside the stack can be of the only static types allowed in Wasm 1.0, `i32` for 32 bits signed integer, `i64` for 64 bits signed integer, `f32` for 32 bits float and `f64` for 64 bits float. Therefore, abstract types, such as classes, objects, and arrays, are not natively supported. Instead, during compilation, such types are transformed into primitive types and stored in the linear memory.

Memory Instances represent the unmanaged linear memory of a WebAssembly program, consisting of a contiguous array of bytes. Memory instances are accessed with `i32` pointers (integer of 32 bits). Memory instances are usually bound in browser engines to 4Gb of size, and it is only shareable between the process that instantiates the WebAssembly module and the binary itself.

Global Instances : A global instance is a global variable with a value and a mutability flag, indicating whether the global can be modified or is immutable. Global variables are part of the managed data, i.e., their allocation and memory placement are managed by the host engine. Global variables are only accessible by their declaration index, and it is not possible to dynamically address them.

Locals : Locals are mutable variables that are local to a specific function instance, i.e. locals are only accessible through their index related to the executing function instance. As globals, locals are part of the managed data.

Function Instances : are closures over the runtime module instance. A function instance groups locals and a function body. Locals are typed variables that are local to a specific function invocation as previously discussed. The function body is a sequence of instructions that are executed when the function is called. Each instruction either reads from the stack, writes to the stack, or modifies the control flow of the function. Recalling the example Wasm binary previously showed, the local variable declarations and typed instructions that are evaluated using the stack can be appreciated between Line 7 and Line 32 in Listing 2.2. Each instruction reads its operands from the stack and pushes back the result. In the case of Listing 2.2, the result value of the main function is the calculation of the last instruction, `i32.add`. As the listing also shows, instructions are annotated with a numeric type.

Definition 1. *Along with this dissertation, as the work of Lehmann et al. [**?**], we refer to managed and unmanaged data to differentiate between the data that is managed by the host engine and the data that is managed by the WebAssembly program respectively.*

2.1.4 WebAssembly's control flow

In WebAssembly, a defined function instructions are organized into blocks, with the function's starting point serving as the root block. Unlike traditional assembly code, control flow structures in Wasm jump between block boundaries rather than arbitrary positions within the code. Each block might specify the required stack state before execution and the resulting stack state after its instructions have run. This stack state is used to validate the binary during compilation and to ensure that the stack is in a valid state before executing the block's instructions. Blocks in Wasm are explicit, indicating, where they start and end. By design, each block cannot reference or execute code from outer blocks.

Control flow within a function is managed through three types of break instructions: unconditional break, conditional break, and table break. Importantly, each break instruction is limited to jumping to one of its enclosing blocks. Unlike standard blocks, where breaks jump to the end of the block, breaks within a loop block jump to the block's beginning, effectively restarting the loop. To illustrate this, Listing 2.3 provides an example comparing a standard block and a loop block in a Wasm function.

```
block
   block
      br 1    ; Jump instructions
            are annotated with the
            depth of the block they
             jump to;
   end
end
...
```

```
loop
   ...
   br 0    ;first-order break;
   ...
end    ; end instructions break
      the block and jump to next
      instruction;
...
```

Listing 2.3: *Example of breaking a block and a loop in WebAssembly.*

Each break instruction includes the depth of the enclosing block as an operand. This depth is used to identify the target block for the break instruction. For example, in the left-most part of the previously discussed listing, a break instruction with a depth of 1 would jump past two enclosing blocks.

2.1.5 WebAssembly's ecosystem

WebAssembly programs are tailored for execution in host environments, most notably web browsers. The WebAssembly ecosystem is a diverse landscape, featuring a multitude of stakeholders and a comprehensive suite of tools to meet various requirements [**?**]. In this section, we delineate two key categories of tools within this ecosystem: compilers and executors. Compilers are responsible for converting source code into WebAssembly binaries, while executors handle a range of tasks including validation, optimization, machine code transpilation, and actual execution of these WebAssembly binaries. Executors are often found in browser clients, among other platform

Compilers transform source code into WebAssembly binaries. For example, LLVM has offered WebAssembly as a backend option since its 7.1.0 release[5], supporting a diverse set of frontend languages like C/C++, Rust, Go, and AssemblyScript[6]. Significantly, a study by Hilbig et al. reveals that 70% of WebAssembly binaries are generated using LLVM-based compilers. In parallel developments, the KMM framework[7] has incorporated WebAssembly as a compilation target, and the Javy approach[8] focuses on encapsulating JavaScript code within isolated WebAssembly binaries. This latter is achieved by porting both the engine and the source code into a secure WebAssembly environment. Similarly, Blazor also enables the compilation of C code into WebAssembly binaries for browser execution[9].

[5] `https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0`

[6] A subset of the TypeScript language

[7] `https://kotlinlang.org/docs/wasm-overview.html`

[8] `https://github.com/bytecodealliance/javy`

[9] `https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor`

From a security standpoint, WebAssembly programs are designed without a standard library and are prohibited from direct interactions with the operating system. Instead, the host environment offers a predefined set of functions that can be imported into the WebAssembly program. It falls upon the compilers to specify which functions from the host environment will be imported by the WebAssembly application.

Browser engines like V8[10] and SpiderMonkey[11] are at the forefront of executing WebAssembly binaries in browser clients. These engines leverage Just-In-Time (JIT) compilers to convert WebAssembly into machine code. This translation is typically a straightforward one-to-one mapping, given that WebAssembly is already an optimized format closely aligned with machine code, as previously discussed in Subsection 2.1.2. For example, V8 just employs quick, rudimentary optimizations, such as constant folding and dead code removal, to guarantee fast readiness for a Wasm binary to execute [**?**].

Standalone engines: Wasm has expanded beyond browser environments, largely due to the WASI[**?**]. It standardizes the interactions between host environments and WebAssembly modules through a POSIX-like interface. Wasm compilers can generate binaries that use WASI. Standalone engines can then execute these binaries in a variety of environments, including cloud, server, and IoT devices. For example, standalone engines like WASM3[12], Wasmer[13], Wasmtime[14], WAVM[15], and Sledge[**?**] have emerged to support WebAssembly and WASI. In a similar vein, Singh et al.[**?**] introduced a virtual machine for WebAssembly tailored for Arduino-based devices. Salim et al.[**?**] proposed TruffleWasm, an implementation of WebAssembly hosted on Truffle and GraalVM. Additionally, SWAM[16] stands out as WebAssembly interpreter implemented in Scala. Finally, WaVe[**?**] offers a WebAssembly interpreter featuring mechanized verification of the WebAssembly-WASI interaction with the underlying operating system.

2.1.6 WebAssembly's binary analysis

As the WebAssembly ecosystem continues to grow, the need for robust tools to ensure its security and reliability has increased. To address this, a variety of tools have been developed that employ different strategies to identify vulnerabilities in WebAssembly programs. In the following text we provide a brief overview of the

[10] `https://chromium.googlesource.com/v8/v8.git`
[11] `https://spidermonkey.dev/`
[12] `https://github.com/wasm3/wasm3`
[13] `https://wasmer.io/`
[14] `https://github.com/bytecodealliance/wasmtime`
[15] `https://github.com/WAVM/WAVM`
[16] `https://github.com/satabin/swam`

most relevant tools in this space w.r.t static and dynamic analysis, as well as specialized malware detection.

Static and dynamic analysis: Tools like Wassail[**?**], SecWasm[**?**], Wasmati[**?**], and Wasp[**?**] leverage techniques such as information flow control, code property graphs, control flow analysis, and concolic execution to detect vulnerabilities in Wasm binaries. Remarkably, VeriWasm[**?**] stands out as a static offline verifier specifically designed for native x86-64 binaries compiled from WebAssembly. In the dynamic analysis counterpart, tools like TaintAssembly[**?**], Wasabi[**?**], and Fuzzm[**?**] offer similar functionalities in vulnerability detection. Stiévenart and colleagues have introduced a dynamic approach to slice WebAssembly programs based on Observational-Based Slicing (ORBS)[**?** **?**]. Hybrid methods have also gained traction, with tools like CT-Wasm[**?**] enabling the verifiably secure implementation of cryptographic algorithms in WebAssembly.

Specialized Malware Detection: Cryptomalwares have a wide presence in the web since the first days of Wasm. The main reason is that mining algorithms using CPUs moved to Wasm for obvious performance reasons [**?**]. In cryptomalware detection, tools like MineSweeper[**?**], MinerRay[**?**], and MINOS[**?**] utilize static analysis through machine learning techniques to detect browser cryptomalwares. Conversely, tools like SEISMIC[**?**], RAPID[**?**], and OUTGuard[**?**] seek the same goal with dynamic analysis techniques. Remarkably, VirusTotal[17], packaging more than 60 commercial antivirus as back-boxes, detects cryptomalware in Wasm binaries.

2.1.7 WebAssembly's security

While WebAssembly is engineered to be deterministic, well-typed, and to adhere to a structured control flow, the ecosystem is still emerging and faces various security vulnerabilities. These vulnerabilities pose risks to both the consumers and the WebAssembly binaries themselves. Side-channel attacks, in particular, are a significant concern. For example, Genkin et al. have shown that WebAssembly can be exploited to exfiltrate data through cache timing-side channels [**?**]. Similarly, research by Maisuradze and Rossow demonstrates the feasibility of speculative execution attacks on WebAssembly binaries [**?**]. Rokicki et al. further reveal the potential for port contention side-channel attacks on WebAssembly binaries in browsers [**?**]. Additionally, studies by Lehmann et al. and Stiévenart and colleagues indicate that vulnerabilities in C/C++ source code can propagate into WebAssembly binaries [**?** **?**]. This dissertation introduces

[17] `https://www.virustotal.com`

a comprehensive set of tools aimed at preemptively enhancing WebAssembly security through Software Diversification and at improving testing rigor within the ecosystem.

2.2 Software diversification

TODO Work on differential testing `https://arxiv.org/pdf/2309.12167.pdf`

2.2.1 Generating Software Diversification

2.2.2 Variants generation

2.2.3 Variants equivalence

TODO Automatic, SMT based **TODO** Take a look to Jackson thesis, we have a similar problem he faced with the superoptimization of NaCL **TODO** By design **TODO** Introduce the notion of rewriting rule by Sasnaukas. `https://link.springer.com/chapter/10.1007/978-3-319-68063-7_13`

2.2.4 Defensive Diversification

2.2.5 Offensive Diversification

3 AUTOMATIC SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

THE process of generating WebAssembly binaries starts with the original source code, which is then processed by a compiler to produce a WebAssembly binary. This compiler is generally divided into three main components: a frontend that converts the source code into an intermediate representation, an optimizer/transformer that modifies this representation usually for performance, and a backend that compiles the final WebAssembly binary. This architecture is illustrated in the left most part of Figure 3.1.

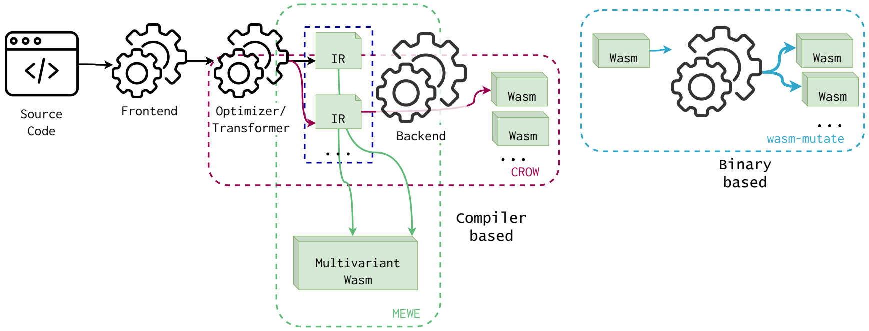

Figure 3.1: *Approach landscape containing our three technical contributions: CROW squared in red, MEWE squared in green and WASM-MUTATE squared in blue. We annotate where our contributions, compiler-based and binary-based, stand in the landscape of generating WebAssembly programs.*

[0] Comp. time 2023/10/06 12:22:33

Software Diversification, a preemptive security measure, can be integrated at various stages of this compilation process. However, applying diversification at the front-end has its limitations, as it would need a unique diversification mechanism for each language compatible with the frontend component. Conversely, diversification at later compiler stages, such as the optimizer or backend, offers a more practical alternative. This makes the latter stages of the compilers an ideal point for introducing practical Wasm diversification techniques. Our compiler-based strategies, represented in red and green in Figure 3.1, introduce a diversifier component into the optimizer/transformer and backend stages. This optimization/transformer component generates variants in the intermediate representation of a compiler, thereby creating artificial software diversity for WebAssembly. The variants are then compiled into WebAssembly binaries by the backend component of the compiler. Specifically, we propose two tools: CROW, which generates WebAssembly program variants, and MEWE, which packages these variants to enable multivariant execution [**?**]. Alternatively, diversification can be directly applied to the WebAssembly binary, offering a language and compiler-agnostic approach. Our binary-based strategy, WASM-MUTATE, represented in blue in Figure 3.1, employs rewriting rules on an e-graph data structure to generate a variety of WebAssembly program variants.

This dissertation contributes to the field of Software Diversification for WebAssembly by presenting two primary strategies: compiler-based and binary-based. Within this chapter, we introduce three technical contributions: CROW, MEWE, and WASM-MUTATE. We also compare these contributions, highlighting their complementary nature. Additionally, we provide the artifacts for our contributions to promote open research and reproducibility of our main takeaways.

3.1 CROW: Code Randomization of WebAssembly

This section details CROW [**?**], represented as the red squared tooling in Figure 3.1. CROW is designed to produce functionally equivalent Wasm variants from the output of an LLVM front-end, utilizing a custom Wasm LLVM backend.

Figure 3.2 illustrates CROW's workflow in generating program variants, a process compound of two core stages: *exploration* and *combination*. During the *exploration* stage, CROW processes every instruction within each function of the LLVM input, creating a set of functionally equivalent code variants. This process ensures a rich pool of options for the subsequent stage. In the *combination* stage, these alternatives are assembled to form diverse LLVM IR variants, a task achieved through the exhaustive traversal of the power set of all potential combinations of code replacements. The final step involves the custom Wasm LLVM backend, which compiles the crafted LLVM IR variants into Wasm binaries.

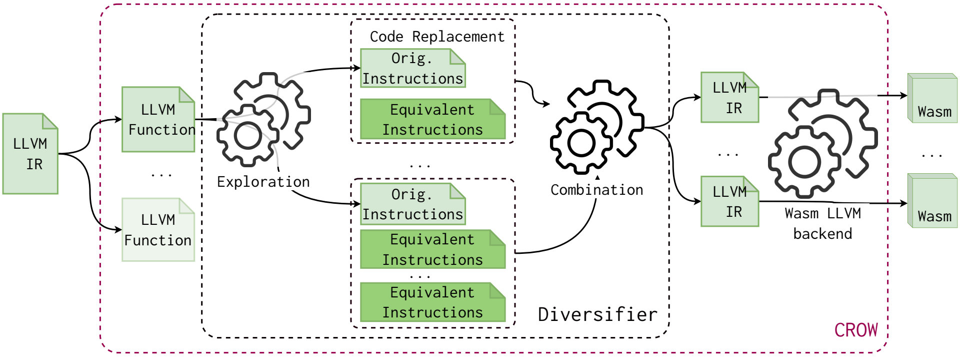

Figure 3.2: *CROW components following the diagram in Figure 3.1. CROW takes LLVM IR to generate functionally equivalent code replacements. Then, CROW assembles program variants by combining them. Figure taken from [?].*

3.1.1 Enumerative synthesis

The cornerstone of CROW's exploration mechanism is its code replacement generation strategy, which is inspired by the superdiversifier methodology proposed by Jacob et al. [?]. The search space for generating variants is delineated through an enumerative synthesis process, which systematically produces all possible code replacements for each instruction and its data flow graph in the original program. If a code replacement is identified to perform identically to the original program, it is reported as a functionally equivalent variant. This equivalence is confirmed using a theorem solver for rigorous verification.

Concretely, CROW is developed by extending the enumerative synthesis implementation found in Souper [?], an LLVM-based superoptimizer. Specifically, CROW constructs a Data Flow Graph for each LLVM instruction that returns an integer. Subsequently, it generates all viable expressions derived from a selected subset of the LLVM Intermediate Representation language for each DFG. The enumerative synthesis process incrementally generates code replacements, starting with the simplest expressions (those composed of a single instruction) and gradually increasing in complexity. The exploration process continues either until a timeout occurs or the size of the generated replacements exceeds a predefined threshold.

Notice that the search space increases exponentially with the size of the language used for enumerative synthesis. To mitigate this issue, we prevent CROW from synthesizing instructions without correspondence in the Wasm backend, effectively reducing the searching space. For example, creating an expression having the `freeze` LLVM instructions will increase the searching space for instruction without a Wasm's opcode in the end.

CROW is carefully designed to boost the generation of variants as much as possible. First, we disable the majority of the pruning strategies. Instead of preventing the generation of commutative operations during the searching, CROW still uses such transformation as a strategy to generate program variants. Second, CROW applies code transformations independently. For instance, if a suitable replacement is identified that can be applied at N different locations in the original program, CROW will generate 2^N distinct program variants, i.e., the power set of applying the transformation or not to each location. This approach leads to a combinatorial explosion in the number of available program variants, especially as the number of possible replacements increases.

Leveraging the ascending nature of its enumerative synthesis process, CROW is capable of creating variants that may outperform the original program in both size and efficiency. For instance, the first functionally equivalent transformation identified is typically the most optimal in terms of code size. This approach offers developers a range of performance options, allowing them to balance between diversification and performance without compromising the latter.

The last stage at CROW involves a custom Wasm LLVM backend, which generates the Wasm programs. For it, we remove all built-in optimizations in the LLVM backend that could reverse Wasm variants, i.e., we disable all optimizations in the Wasm backend that could reverse the CROW transformations.

3.1.2 Constant inferring

CROW inherently introduces a novel transformation strategy called *constant inferring*, which significantly expands the variety of WebAssembly program variants. Specifically, CROW identifies segments of code that can be simplified into a single constant assignment, with a particular focus on variables that control branching logic. After applying this *constant inferring* technique, the resulting program diverges substantially from the original program structure. This is crucial for diversification efforts, as one of the primary objectives is to create variants that are as distinct as possible from the original source code [**?**]. In essence, the more divergent the variant, the more challenging it becomes to trace it back to its original form.

Let us illustrate the case with an example. The Babbage problem code in Listing 3.1 is composed of a loop that stops when it discovers the smallest number that fits with the Babbage condition in Line 4.

```
1   int babbage() {
2       int current = 0,
3           square;
4       while ((square=current*current) %
              ↪ 1000000 != 269696) {
5           current++;
6       }
7       printf ("The number is %d\n",
              ↪ current);
8       return 0 ;
9   }
```

Listing 3.1: *Babbage problem. Taken from [?].*

```
int babbage() {
    int current = 25264;





    printf ("The number is %d\n", current)
        ↪ ;
    return 0 ;
}
```

Listing 3.2: *Constant inferring transformation over the original Babbage problem in Listing 3.1. Taken from [?].*

CROW deals with this case, generating the program in Listing 3.2. It infers the value of `current` in Line 2 such that the Babbage condition is reached [1]. Therefore, the condition in the loop will always be false. Then, the loop is dead code and is removed in the final compilation. The new program in Listing 3.2 is remarkably smaller and faster than the original code. Therefore, it offers differences both statically and at runtime[2]

3.1.3 Exemplifying CROW

Let us illustrate how CROW works with the example code in Listing 3.3. The `f` function calculates the value of $2 * x + x$ where `x` is the input for the function. CROW compiles this source code and generates the intermediate LLVM bitcode in the left most part of Listing 3.4. CROW potentially finds two integer returning instructions to look for variants, as the right-most part of Listing 3.4 shows.

```
int f(int x) {
    return 2 * x + x;
}
```

Listing 3.3: *C function that calculates the quantity $2x + x$.*

[1] In theory, this value can also be inferred by unrolling the loop the correct number of times with the LLVM toolchain. However, standard LLVM tools cannot unroll the `while`-loop because the loop count is too large.

[2] Notice that for the sake of illustration, we show both codes in C language, this process inside CROW is performed directly in LLVM IR.

```
define i32 @f(i32) {       Replacement candidates   Replacement candidates for
                                 for code_1                code_2
  %2 = mul nsw i32 %0,2
  %3 = add nsw i32 %0,%2   %2 = mul nsw i32 %0,2     %3 = add nsw i32 %0,%2

  ret i32 %3               %2 = add nsw i32 %0,%0    %3 = mul nsw %0, 3:i32
}
                           %2 = shl nsw i32 %0, 1:i32
define i32 @main() {
  %1 = tail call i32 @f(
        i32 10)
  ret i32 %1
}
```

Listing 3.4: *LLVM's intermediate representation program, its extracted instructions and replacement candidates. Gray highlighted lines represent original code, green for code replacements.*

```
%2 = mul nsw i32 %0,2              %2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2             %3 = mul nsw %0, 3:i32

%2 = add nsw i32 %0,%0             %2 = add nsw i32 %0,%0
%3 = add nsw i32 %0,%2             %3 = mul nsw %0, 3:i32

%2 = shl nsw i32 %0, 1:i32         %2 = shl nsw i32 %0, 1:i32
%3 = add nsw i32 %0,%2             %3 = mul nsw %0, 3:i32
```

Listing 3.5: *Candidate code replacements combination. Orange highlighted code illustrate replacement candidate overlapping.*

CROW, detects `code_1` and `code_2` as the enclosing boxes in the left most part of Listing 3.4 shows. CROW synthesizes $2+1$ candidate code replacements for each code respectively as the green highlighted lines show in the right most parts of Listing 3.4. The baseline strategy of CROW is to generate variants out of all possible combinations of the candidate code replacements, *i.e.,* uses the power set of all candidate code replacements.

In the example, the power set is the cartesian product of the found candidate code replacements for each code block, including the original ones, as Listing 3.5 shows. The power set size results in 6 potential function variants. Yet, the generation stage would eventually generate 4 variants from the original program. CROW generated 4 statically different Wasm files, as Listing 3.6 illustrates. This gap between the potential and the actual number of variants is a consequence of the redundancy among the bitcode variants when composed into one. In other words, if the replaced code removes other code blocks, all possible combinations having it will be in the end the same program. In the example case, replacing `code_2` by `mul nsw %0, 3`, turns `code_1` into dead code, thus, later replacements generate the same program variants. The rightmost part of Listing 3.5 illustrates how for three different combinations, CROW produces the same variant. We call this phenomenon a *code replacement overlapping.*

```
func $f (param i32) (result i32)
   local.get 0
    i32.const 2
    i32.mul
    local.get 0
    i32.add
```

```
func $f (param i32) (result i32)
    local.get 0
    i32.const 1
    i32.shl
    local.get 0
    i32.add
```

```
func $f (param i32) (result i32)
    local.get 0
    local.get 0
    i32.add
    local.get 0
    i32.add
```

```
func $f (param i32) (result i32)
    local.get 0
    i32.const 3
    i32.mul
```

Listing 3.6: *Wasm program variants generated from program Listing 3.3.*

Contribution paper and artifact

CROW is a compiler-based approach. It leverages enumerative synthesis to generate functionally equivalent code replacements and assembles them into diverse Wasm program variants. CROW uses SMT solvers to guarantee functional equivalence.
CROW is fully presented in Cabrera-Arteaga et al. "CROW: Code Randomization of WebAssembly" *at proceedings of Measurements, Attacks, and Defenses for the Web (MADWeb), NDSS 2021* `https://doi.org/10.14722/madweb.2021.23004`.

CROW source code is available at `https://github.com/ASSERT-KTH/slumps`

3.2 MEWE: Multi-variant Execution for WebAssembly

This section describes MEWE [**?**]. MEWE synthesizes diversified function variants by using CROW. It then provides execution-path randomization in a Multivariant Execution (MVE) [**?**]. Execution path randomization is a technique that randomizes the execution path of a program at runtime, i.e. at each invocation of a function, a different variant is executed [**?**]. MEWE generates application-level multivariant binaries without changing the operating system or Wasm runtime. It creates an MVE by intermixing functions for which CROW generates variants, as illustrated by the green square in Figure 3.1. MEWE inlines function variants when appropriate, resulting in call stack diversification at runtime.

As illustrated in Figure 3.3, MEWE takes the LLVM IR variants generated by CROW's diversifier. It then merges LLVM IR variants into a Wasm multivariant. In the figure, we highlight the two components of MEWE, *Multivariant Generation* and the *Mixer*. In the *Multivariant Generation* process, MEWE gathers the LLVM IR variants created by CROW. The Mixer component, on the other hand, links the multivariant binary and creates a new entrypoint for the binary called *entrypoint tampering*. The tampering is needed in case the output of CROW are variants of the original entrypoint, e.g. the *main* function. Concretely, it wraps the dispatcher for the entrypoint variants as a new function for the final Wasm binary and is declared as the application entrypoint. The random generator is needed to perform the execution-path randomization. For the random generator, we rely on WASI's specification [**?**] for the random behavior of the dispatchers. However, its exact implementation is dependent on the platform on which the binary is deployed. Finally, using the same custom Wasm LLVM backend as CROW, we generate a standalone multivariant Wasm binary. Once generated, the multivariant Wasm binary can be deployed to any Wasm engine.

3.2.1 Multivariant call graph

The key component of MEWE consists of combining the variants into a single binary. The core idea is to introduce one dispatcher function per original function with variants. A dispatcher function is a synthetic function in charge of choosing a variant at random when the original function is called. With the introduction of the dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

Definition 2. *Multivariant Call Graph (MCG): A multivariant call graph is a call graph $\langle N, E\rangle$ where the nodes in N represent all the functions in the binary and an edge $(f_1, f_2) \in E$ represents a possible invocation of f_2 by f_1 [**?**]. The nodes in N have three possible types: a function present in the original program, a generated function variant, or a dispatcher function.*

3.2.2 Exemplifying a Multivariant binary

In Figure 3.4, we show the original static call graph for an original program (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The gray nodes represent function variants, the green nodes function dispatchers, and the yellow nodes are the original functions. The directed edges represent the possible calls. The original program includes three functions. MEWE generates 43 variants for the first function, none for the second, and three for the third. MEWE introduces two dispatcher nodes for the first and third functions. Each dispatcher is connected to the corresponding function variants to invoke one variant randomly at runtime.

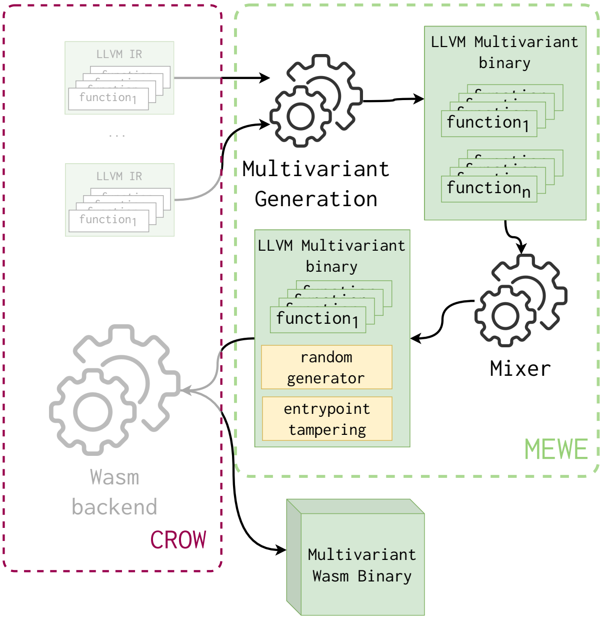

Figure 3.3: *Overview of MEWE workflow. It takes as input an LLVM binary. It first generates a set of functionally equivalent variants for each function in the binary using CROW. Then, MEWE generates an LLVM multivariant binary composed of all the function variants. Finally, the Mixer includes the behavior in charge of selecting a variant when a function is invoked. Finally, the MEWE mixer composes the LLVM multivariant binary with a random number generation library and tampers the original application entrypoint. The final process produces a Wasm multivariant binary ready to be deployed. Figure partially taken from [**?**].*

In Listing 3.7, we demonstrate how MEWE constructs the function dispatcher, corresponding to the rightmost green node in Figure 3.4, which handles three created variants including the original. The dispatcher function retains the same signature as the original function. Initially, the dispatcher invokes a random number generator, the output of which is used to select a specific function variant for execution (as seen on line 6 in Listing 3.7). To enhance security, we employ a switch-case structure within the dispatcher, mitigating vulnerabilities associated with speculative execution-based attacks [**?**] (refer to lines 12 to 19 in Listing 3.7). This approach also eliminates the need for multiple function definitions with identical signatures, thereby reducing the potential attack surface in cases where the function signature itself is vulnerable [**?**]. Additionally, MEWE can inline function variants directly into the dispatcher, obviating the need for redundant definitions (as illustrated on line 16 in Listing 3.7). Remarkably, we prioritize security over performance, i.e., while using indirect

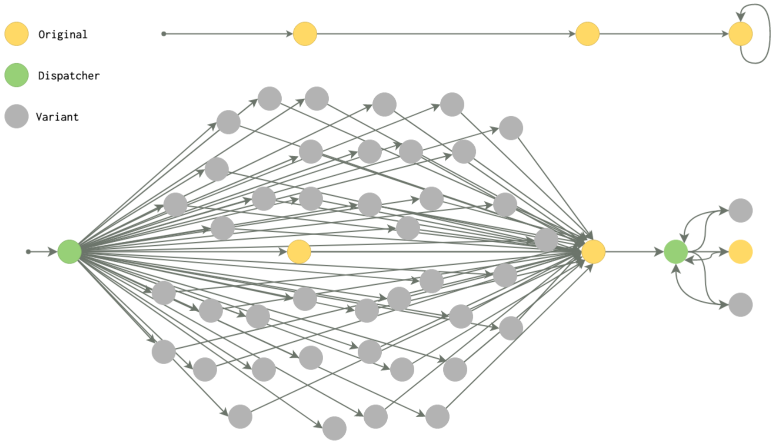

Figure 3.4: *Example of two static call graphs. At the top, is the original call graph, and at the bottom, is the multivariant call graph, which includes nodes that represent function variants (in gray), dispatchers (in green), and original functions (in yellow). Figure taken from [**?**].*

calls in place of a switch-case could offer constant-time performance benefits, we implement switch-case structures.

```
2   ; Multivariant foo wrapping ;
3   define internal i32 @foo(i32 %0) {
4       entry:
5         ; It first calls the dispatcher to discriminate between the created
              variants ;
6         %1 = call i32 @discriminate(i32 3)
7         switch i32 %1, label %end [
8           i32 0, label %case_43_
9           i32 1, label %case_44_
10        ]
11      ;One case for each generated variant of foo ;
12      case_43_:
13        %2 = call i32 @foo_43_(%0)
14        ret i32 %2
15      case_44_:
16        ; MEWE can inline the body of the a function variant ;
17        %3 = <body of foo_44_ inlined>
18        ret i32 %3
19      end:
20        ; The original is also included ;
21        %4 = call i32 @foo_original(%0)
22        ret i32 %4
23  }
```

Listing 3.7: *Dispatcher function embedded in the multivariant binary of the original function in the rightmost green node in Figure 3.4. The code is commented for the sake of understanding.*

In Listing 3.7, we illustrate the LLVM construction for the function dispatcher corresponding to the right most green node of Figure 3.4. Notice that, the dispatcher function is constructed using the same signature as the original function. It first calls the random generator, which returns a value used to invoke a specific function variant (see line 6 in Listing 3.7). We utilize a switch-case structure in the dispatchers to prevent indirect calls, which are vulnerable to speculative execution-based attacks [?] (see lines 12 to 19 in Listing 3.7), i.e., the choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [?]. In addition, MEWE can inline function variants inside the dispatcher instead of defining them again (see line 16 in Listing 3.7). Remarkably, we trade security over performance since dispatcher functions that perform indirect calls, instead of a switch-case, could improve the performance of the dispatchers as indirect calls have constant time.

Contribution paper and artifact

MEWE provides dynamic execution path randomization by packaging variants generated out of CROW.
MEWE is fully presented in Cabrera-Arteaga et al. "Multi-Variant Execution at the Edge" *Proceedings of Moving Target Defense, 2022, ACM* `https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

MEWE is also available as an open-source tool at `https://github.com/ASSERT-KTH/MEWE`

3.3 WASM-MUTATE: Fast and Effective Binary for WebAssembly

In this section, we introduce our third technical contribution, WASM-MUTATE [?], a tool that generates thousands of functionally equivalent variants out from a WebAssembly binary input. Leveraging rewriting rules and e-graphs [?] for software diversification, WASM-MUTATE synthesizes program variants by transforming parts of the original binary. In Figure 3.1, we highlight WASM-MUTATE as the blue squared tooling.

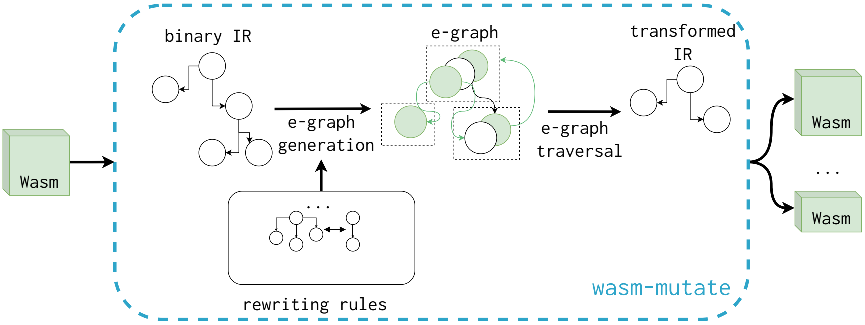

Figure 3.5: *WASM-MUTATE high-level architecture. It generates functionally equivalent variants from a given WebAssembly binary input. Its central approach involves synthesizing these variants by substituting parts of the original binary using rewriting rules, boosted by diversification space traversals using e-graphs.*

Figure 3.5 illustrates the workflow of WASM-MUTATE, which initiates with a WebAssembly binary as its input. The first step involves parsing this binary to create suitable abstractions, e.g. an intermediate representation. Subsequently, WASM-MUTATE utilizes predefined rewriting rules to construct an e-graph for the initial program, encapsulating all potential equivalent codes derived from the rewriting rules. The assurance of functional equivalence is rooted in the inherent properties of the individual rewrite rules employed. Then, pieces of the original program are randomly substituted by the result of random e-graph traversals, resulting in a variant that maintains functional equivalence to the original binary. WASM-MUTATE applies one transformation at a time. The output of one applied transformation can be chained again as an input WebAssembly binary, enabling the generation of many variants.

3.3.1 WebAssembly Rewriting Rules

WASM-MUTATE contains a comprehensive set of 135 rewriting rules. In this context, a rewriting rule is a tuple `(LHS, RHS, Cond)` where `LHS` specifies the segment of binary targeted for replacement, `RHS` describes its functionally equivalent substitute, and `Cond` outlines the conditions that must be met for the replacement to take place, e.g. enhancing type constraints. WASM-MUTATE groups these rewriting rules into meta-rules depending on their target inside a Wasm binary, ranging from high-level changes affecting binary section structure to low-level modifications within the code section. This section focuses on the biggest meta-rule implemented in WASM-MUTATE, the `Peephole` meta-rule [3].

Rewriting rules inside the *Peephole* meta-rule, operate over the data flow graph of instructions within a function body, representing the lowest level of rewriting. In WASM-MUTATE, we have implemented 125 rewriting rules specifically for this category, each one avoiding targeting instructions that might induce undefined behavior, e.g., function calls.

Moreover, we augment the internal representation of a Wasm program to bolster WASM-MUTATE's transformation capabilities through the `Peephole` meta-rule. Concretely, we augment the parsing stage in WASM-MUTATE by including custom operator instructions. These custom operator instructions are designed to use well-established code diversification techniques through rewriting rules. When converting back to the WebAssembly binary format from the intermediate representation, custom instructions are meticulously handled to retain the original functionality of the WebAssembly program.

In the following example, we demonstrate a rewriting rule within the `Peephole` meta-rule that utilizes a custom `rand` operator to expand statically declared constants within any WebAssembly program function body. The unfold rewriting rule, as the name suggests, transforms statically declared constants into the sum of two random numbers. During the generation of the WebAssembly variant, the custom `rand` operator is substituted with a randomly chosen static constant. Notice that the condition specified in the last part of the rewriting rule ensures that this predicate is satisfied.

[3] For an in-depth explanation of the remaining meta-rules, refer to [?].

```
    LHS i32.const x
----------------------------------------
    RHS (i32.add (i32.rand i32.const y))

Cond y = x - i32.rand
```

Although this rewriting approach may appear simplistic, especially because compilers often eliminate it through *Constant Folding* optimization [?], it stresses on the spill/reload component of the compiler when the WebAssembly binary is transpiled to machine code. Spill/reloads occur when the compiler runs out of physical registers to store intermediate calculations, resorting to specific memory locations for temporary storage. The unfold rewriting rule indirectly stresses this segment of memory. Notably, with this specific rewriting rule, we have found a CVE in the wasmtime standalone engine [?].

3.3.2 E-Graphs traversals

We developed WASM-MUTATE leveraging e-graphs, a specific graph data structure for representing and applying rewriting rules [?]. In the context of WASM-MUTATE, e-graphs are constructed from the input WebAssembly program and the implemented rewriting rules (we detail the e-graph construction process in Section 3 of [?]).

Willsey et al. highlight the potential for high flexibility in extracting code fragments from e-graphs, a process that can be recursively orchestrated through a cost function applied to e-nodes and their respective operands. This methodology ensures the functional equivalence of the derived code [?]. For instance, e-graphs solve the problem of providing the best code out of several optimization rules [?]. To extract the "optimal" code from an e-graph, one might commence the extraction at a specific e-node, subsequently selecting the AST with the minimal size from the available options within the corresponding e-class's operands. In omitting the cost function from the extraction strategy leads us to a significant property: *any path navigated through the e-graph yields a functionally equivalent code variant.*

We exploit such property to fastly generate diverse WebAssembly variants. We propose and implement an algorithm that facilitates the random traversal of an e-graph to yield functionally equivalent program variants, as detailed in Algorithm 1. This algorithm operates by taking an e-graph, an e-class node (starting with the root's e-class), and a parameter specifying the maximum extraction depth of the expression, to prevent infinite recursion. Within the algorithm, a random e-node is chosen from the e-class (as seen in lines 5 and 6), setting the stage for a recursive continuation with the offspring of the selected e-node (refer to line 8). Once the depth parameter reaches zero, the algorithm extracts the most concise expression available within the current e-class (line

3). Following this, the subexpressions are built (line 10) for each child node, culminating in the return of the complete expression (line 11).

Algorithm 1 e-graph traversal algorithm taken from [**?**].

```
 1: procedure TRAVERSE($egraph$, $eclass$, $depth$)
 2:     if depth = 0 then
 3:         return smallest_tree_from(egraph, eclass)
 4:     else
 5:         $nodes \leftarrow egraph[eclass]$
 6:         $node \leftarrow random\_choice(nodes)$
 7:         $expr \leftarrow (node, operands = [])$
 8:         for each $child \in node.children$ do
 9:             $subexpr \leftarrow$ TRAVERSE($egraph$, $child$, $depth - 1$)
10:             $expr.operands \leftarrow expr.operands \cup \{subexpr\}$
11:         return $expr$
```

3.3.3 Exemplifying WASM-MUTATE

Let us illustrate how WASM-MUTATE generates variant programs by using the before enunciated algorithm. Here, we use Algorithm 1 with a maximum depth of 1. In Listing 3.8 a hypothetical original Wasm binary is illustrated. In this context, a potential user has set two pivotal rewriting rules: `(x, container (x nop),)` and `(x, x i32.add 0, x instanceof i32)`. The former rule, grants the ability to append a `nop` instruction to any subexpression, a well-known low-level diversification strategy [**?**]. Conversely, the latter rule adds zero to any numeric value.

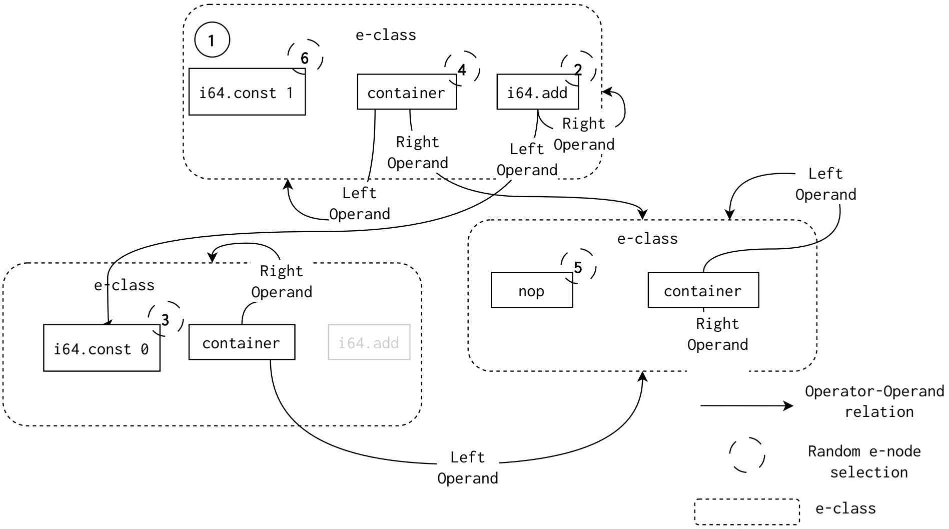

Figure 3.6: e-graph built for rewriting the first instruction of Listing 3.8.

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        i64.const 1)
)
```

Listing 3.8: Wasm function.

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        (i64.add (
            i64.const 0
            i64.const 1
            nop
        ))
    )
```

Listing 3.9: Random peephole mutation using egraph traversal for Listing 3.8 over e-graph Figure 3.6. The textual format is folded for better understanding.

Leveraging the code presented in Listing 3.8 alongside the defined rewriting rules, we build the e-graph, simplified in Figure 3.6. In the figure, we highlight various stages of Algorithm 1 in the context of the scenario previously described. The algorithm initiates at the e-class with the instruction `i64.const 1`, as seen in Listing 3.8. At ②, it randomly selects an equivalent node within the e-class, in this instance taking the `i64.add` node, resulting: `expr`

`= i64.add l r`. As the traversal advances, it follows on the left operand of the previously chosen node, settling on the `i64.const 0` node within the same e-class ③. Then, the right operand of the `i64.add` node is choosen, selecting the `container` ④ operator yielding: `expr = i64.or (i64.const 0 container ( r nop ))`. The algorithm chooses the right operand of the `container` ⑤, which correlates to the initial instruction e-node highlighted in ⑥, culminating in the final expression: `expr = i64.or (i64.const 0 container( i64.const 1 nop)) i64.const 1`. As we proceed to the encoding phases, the `container` operator is ignored as a real Wasm instruction, finally resulting in the program in Listing 3.9.

Notice that, within the e-graph showcased in Figure 3.6, the container node maintains equivalence across all e-classes. Consequently, increasing the depth parameter in Algorithm 1 would potentially escalate the number of viable variants infinitely.

Contribution paper and artifact

WASM-MUTATE uses hand-made rewriting rules and random traversals over e-graphs to provide a binary-based solution for WebAssembly diversification.
WASM-MUTATE is fully presented in Cabrera-Arteaga et al. "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly" *Under review at Computers & Security* `https://arxiv.org/pdf/2309.07638.pdf`.

WASM-MUTATE is available at `https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-mutate` as a contribution to the Bytecode Alliance organization [a]. The Bytecode Alliance is dedicated to creating secure new software foundations, building on standards such as WebAssembly and WASI.

[a] `https://bytecodealliance.org/`

3.4 Comparing CROW, MEWE, and WASM-MUTATE

In this section, we compare CROW, MEWE, and WASM-MUTATE, highlighting their key differences. These distinctions are summarized in Table 3.1. The table is organized into columns that represent attributes of each tool: the tool's name, input format, core diversification strategy, number of variants generated within an hour, targeted sections of the WebAssembly binary for diversification, strength of the generated variants, and the security applications of these variants. Each row in the table corresponds to a specific tool. The *Variant strength* accounts for the capability of each tool on generating variants that are preserved after the JIT compilation of V8 and wasmtime in average. For example, a higher value of the *Variant strength* indicates that the generated variants are not reversed by JIT compilers, ensuring that the diversification is preserved in an end-to-end scenario of a WebAssembly program, i.e. from the source code to its final execution. Notice that, the data and insights presented in the table are sourced from the respective papers of each tool and, from the previous discussion in this chapter.

CROW is a compiler-based strategy, needing access to the source code or its LLVM IR representation to work. Its core is an enumerative synthesis implementation with functionallity verification using SMT solvers, ensuring the functional equivalence of the generated variants. In addition, MEWE extends the capabilities of CROW, utilizing the same underlying technology to create program variants. It goes a step further by packaging the LLVM IR variants into a Wasm multivariant, providing MVE through execution path randomization. Both CROW and MEWE are fully automated, requiring no user intervention besides the input source code. WASM-MUTATE, on the other hand, is a binary-based tool. It uses a set of rewriting rules and the input Wasm binary to generate program variants, centralizing its core around random e-graph traversals. Remarkably, WASM-MUTATE removes the need for compiler adjustments, offering compatibility with any existing WebAssembly binary.

We have observed several interesting phenomena when aggregating the empirical data presented in the corresponding papers of CROW, MEWE and WASM-MUTATE [**?** **?** **?**]. This can be appreciated in the fourth, fifth and sixth columns of Table 3.1. We have observed that WASM-MUTATE generates more unique variants in one hour than CROW and MEWE in at least one order of magnitude. This is mainly because of three reasons. First, CROW and MEWE rely on SMT solvers to prove functionally equivalence, placing a bottleneck when generating variants. Second, CROW and MEWE generation capabilities are limited by the *overlapping* phenomenon discussed in Subsection 3.1.3. Third, WASM-MUTATE can generate variants in any part of the Wasm binary, while CROW and MEWE are limited to the code and function sections.

On the other hand, CROW and MEWE, by using enumerative synthesis, ensure that the generated variants are preserved. In other words, the transformations generated out of CROW and MEWE are virtually irreversible by JIT compilers, such as V8 and wasmtime. This phenomenon is highlighted in the *Variants strength* column of Table 3.1, where we show that CROW and MEWE generate variants with 96% of preservation against 75% of WASM-MUTATE. High preservation is especially important where the preservation of the diversification is crucial, e.g. to hinder reverse engineering.

Tool	**Input**	**Core**	**Variants in 1h**	**Target**	**Variants Strength**	**Security applications**
CROW	Source code or LLVM Ir	Enumerative synthesis with functional equivalence proved through SMT solvers	> 1k	Code section	**96%**	Hinders Static analysis reverse engineering.
MEWE	Source code or LL VM Ir	CROW, Multivariant execution	> 1k	Code and Function sections	**96%**	Hinders, static and dynamic analysis reverse engineering and, web timing-based attacks.
WASM-MUTATE	**Wasm binary**	hand-made rewriting rules, e-graph random traversals	**> 10k**	**Any Wasm section**	76%	Hinders signature-based identification, and cache timing side-channel attacks.

Table 3.1: *Comparing CROW, MEWE and WASM-MUTATE. The table columns are: the tool's name, input format, core diversification strategy, number of variants generated within an hour, targeted sections of the WebAssembly binary, strength of the generated variants, and the security applications of these variants. The* Variant strength *accounts for the capability of each tool on generating variants that are preserved after the JIT compilation of V8 and wasmtime in average. Our three technical contributions are complementary tools that can be combined.*

Takeaway

Our three technical contributions serve as complementary tools that can be combined. For instance, when the source code for a WebAssembly binary is either non-existent or inaccessible, WASM-MUTATE offers a viable solution for generating code variants. On the other hand, CROW and MEWE excel in scenarios where high preservation is crucial.

3.4.1 Security applications

The final column of Table 3.1 emphasizes the security benefits derived from the variants produced by our three key technical contributions. One immediate advantage of altering the structure of WebAssembly binaries across different variants is the mitigation of signature-based identification, thereby enhancing resistance to static reverse engineering. Additionally, our tools generate a diverse array of code variants that are highly preserved. This implies that these variants, each with their unique WebAssembly code, retain their distinct characteristics even after being translated into machine code by JIT compilers. This high level of preservation significantly mitigates the risks associated with side-channel attacks that target specific machine code instructions, such as port contention attacks [**?**]. For instance, if a WebAssembly binary is transformed in such a manner that its resulting machine code instructions differ from the original, it becomes more challenging for a side-channel attack. Conversely, if the compiler translates the variant into machine code that closely resembles the original, the side-channel attack could still exploit those instructions to extract information about the original WebAssembly binary.

Altering the layout of a WebAssembly program inherently influences its managed memory during runtime (see Definition 1). This phenomenon is especially important for CROW and MEWE, given that they do not directly address the WebAssembly memory model. Significantly, CROW and MEWE considerably alter the managed memory by modifying the layout of the WebAssembly program. For example, the *constant inferring* transformations significantly alter the layout of program variants, affecting unmanaged memory elements such as the returning address of a function. Furthermore, WASM-MUTATE not only affects managed memory through changes in the WebAssembly program layout. It also adds rewriting rules to transform unmanaged memory instructions. Memory alterations, either to the unmanaged or managed memories, have substantial security implications, by eliminating potential cache timing side-channels [**?**].

Last but not least, our technical contributions enhance security against web timing-based attacks [**?**] by creating variants that exhibit a wide range of execution times, including faster variants compared to the original program.

This strategy is especially prominent in MEWE's approach, which develops multivariants functioning on randomizing execution paths, thereby thwarting attempts at timing-based inference attacks [**?**]. Adding another layer benefit from MEWE, the integration of diverse variants into multivariants can potentially disrupt dynamic reverse engineering tools such as symbolic executors [**?**]. Concretely, different control flows through a random discriminator, exponentially increase the number of possible execution paths, making multivariant binaries virtually unexplorable.

Takeaway

CROW, MEWE and WASM-MUTATE generate WebAssembly variants that can be used to enhance security. Overall, they generate variants that are suitable for hardening static and dynamic analysis, side-channel attacks, and, to thwart signature-based identification.

3.5 Conclusions

In this chapter, we discuss the technical specifics underlying our primary technical contributions. We elucidate the mechanisms through which CROW generates program variants. Subsequently, we discuss MEWE, offering a detailed examination of its role in forging MVE for WebAssembly. We also explore the details of WASM-MUTATE, proposing a novel e-graph traversal algorithm to fast spawn Wasm program variants. Remarkably, we undertake a comparative analysis of the three tools, highlighting their respective benefits and limitations, alongside the potential security applications of the generated Wasm variants.

In Chapter 4, we present two use cases that support the exploitation of these tools. Chapter 4 serves to bridge theory with practice, showcasing the tangible impacts and benefits realized through the deployment of CROW, MEWE, and WASM-MUTATE.

4 EXPLOITING SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

In this chapter we instantiate the usage of Software Diversification for offensive and defensive purposes. We present two selected use cases that exploit Software Diversification through our technical contributions presented in Chapter 3. The selected cases are representative of applications of Software Diversification for WebAssembly in browsers and standalone engines.

4.1 Offensive Diversification: Malware evasion

The primary malicious use of WebAssembly in browsers is cryptojacking [?]. This is due to the essence of cryptojacking, the faster the mining, the better. Although the research of Lehmann and colleagues [?] suggests a decline in browser-based cryptominers, mainly due to the shutdown of Coinhive, a 2022 report by Kaspersky indicates that the use of cryptominers is on the rise [?]. This underscores the ongoing need for effective automatic detection of cryptojacking malware.

Both antivirus software and browsers have implemented measures to detect cryptojacking. For instance, Firefox employs deny lists to detect cryptomining activities [?]. The academic community has also contributed to the body of work on detecting or preventing WebAssembly-based cryptojacking, as outlined in Subsection 2.1.6. However, it's worth noting that malicious actors can employ evasion techniques to circumvent these detection mechanisms. Bhansali et al. are among the first who have investigated how WebAssembly cryptojacking could potentially evade detection [?], highlighting the critical importance of this use case. For an in-depth discussion on this topic, we direct the reader to our

[0] Comp. time 2023/10/06 12:22:33

contribution [?]. The use of case illustrated in the subsequent sections uses Offensive Software Diversification for the sake of evading malware detection in WebAssembly.
TODO Replace by a time diagram.

4.1.1 Threat model: cryptojacking

Let us illustrate the threat model in which a malicious Wasm binary could be involved. Figure 4.1 illustrates a browser attack scenario: a practical WebAssembly cryptojacking attack consists of three components: a WebAssembly binary, a JavaScript wrapper, and a backend cryptominer pool. The WebAssembly binary is responsible for executing the hash calculations, which consume significant computational resources. The JavaScript wrapper facilitates the communication between the WebAssembly binary and the cryptominer pool.

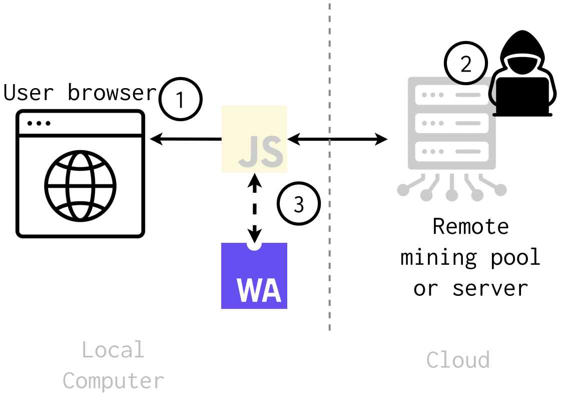

Figure 4.1: *Cryptojacking components.*

For the previous triad to work, the following steps are executed. First, the victim visits a web page infected with the cryptojacking code. The web page establishes a channel to the cryptominer pool, which then assigns a hashing job to the infected browser. The WebAssembly cryptominer calculates thousands of hashes inside the browser. Once the malware server receives acceptable hashes, it is rewarded with cryptocurrencies for the mining. Then, the server assigns a new job, and the mining process starts over.

4.1.2 Approach

Several techniques, as outlined in Subsection 2.1.6, can be directly implemented in browsers to thwart cryptojacking by identifying the malicious WebAssembly components. Such defense scenario is illustrated in Figure 4.2, where the

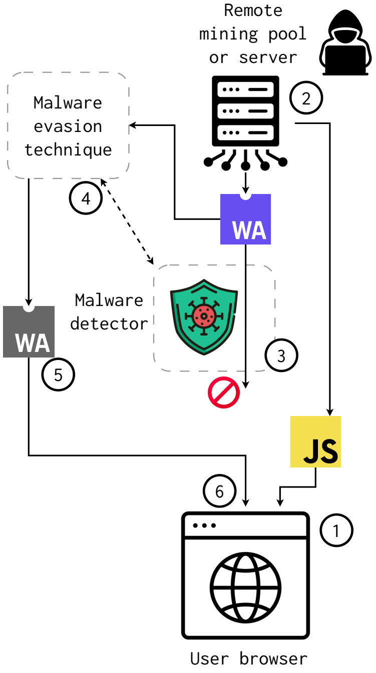

Figure 4.2

WebAssembly binary is blocked in ③. The primary aim of this use of case is to investigate the effectiveness of code diversification as a means to circumvent cryptojacking defenses. Specifically, we assess whether the following evasion workflow can successfully bypass existing security measures:

1. The user lands on a webpage infected with cryptojacking malware, which leverages network resources for execution—corresponding to ① and ② in Figure 4.2. Notice that, various methods can be used to inject cryptojacking malware, including malicious browser extensions, malvertising, compromised websites, or deceptive links [?].

2. A malware detection mechanism identifies and blocks malicious WebAssembly binaries at ③. For example, a network proxy could intercept and forward these resources to an external detection service via its API.

3. Anticipating that a specific malware detection system is consistently used for defense, the attacker swiftly generates a variant of the WebAssembly cryptojacking malware designed to evade detection at ④.

4. The attacker delivers the modified binary instead of the original one ⑤, which initiates the cryptojacking process and compromises the browser ⑥.

The detection method is completely oblivious to the malicious nature of the binary, and the attack is successful.

To empirically validate this evasion scenario, we conducted experiments on 33 cryptojacking malware samples, curated from the 8643 binaries in the wasmbench dataset [?]. These 33 binaries were flagged as potentially hazardous by at least one antivirus vendor on VirusTotal. We employed WASM-MUTATE to simulate ④ in Figure 4.2. Our evaluation focuses on three key metrics: the success rate of evading VirusTotal's detection mechanisms across the 33 binaries, the speed at which WASM-MUTATE generates a detection-evasive variant, and the performance impact on the variants that successfully evade the detection.

4.1.3 Results

TODO Table: Uncontrolled and controlled diversification

TODO How many iterations are needed to evade the detection? Compare with the table in chapter 3, 10000 variants in one hour is equivalent to say that we can evade the detection in 10 minutes.

TODO Table

TODO Talk about performance. Highlight the presence of bloating...the faster you compile, the faster you start ahasing. Link this with the paper that Marin shared lately, highlighting the lack of optimization.

Remarkably, we find 30 cryptominers for which our technique successfully generates variants that evade VirusTotal. Our set of malware includes 6 cryptojacking programs that are fully reproducible in a controlled environment. With them, we assess that our evasion method does not affect malware correctness and generates fully functional malware variants with minimal overhead.

Our work provides evidence that the malware detection community has opportunities to strengthen the automatic detection of cryptojacking WebAssembly malware. The results of this work are actionable, as we provide quantitative evidence on specific malware transformations on which detection methods can focus.

Contribution paper

The case discussed in this section is fully detailed in Cabrera-Arteaga et al. "WebAssembly Diversification for Malware Evasion" *at Computers & Security, 2023* `https://www.sciencedirect.com/science/article/pii/S0167404823002067`.

4.2 Defensive Diversification: Speculative Side-channel protection

As previously discussed in Subsection 2.1.5, WebAssembly is rapidly gaining traction in backend environments. Companies like Cloudflare and Fastly are encouraging the use of WebAssembly in their edge computing platforms, allowing developers to deploy faster applications in a modular and securely sandboxed fashion. Typically, these client WebAssembly applications are designed as isolated services with a single, focused responsibility. This model is usually called Function-as-a-Service (FaaS) [? ?]. In Figure 4.3 we illustrate how WebAssembly binaries are used in FaaS platforms. Developers can submit any WebAssembly binary to the FaaS platform that are then replicated and moved around the Edge-cloud worldwide platform.

TODO Align the figure with the terms

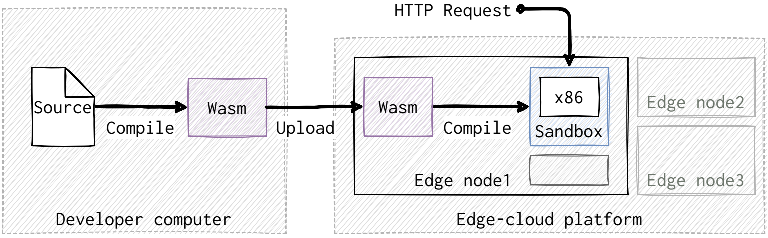

Figure 4.3: WebAssembly binaries on FaaS platforms.

The core concept behind Wasm in FaaS platforms is the ability to host thousands of client WebAssembly binaries within a single host process, which is then distributed across multiple servers and data centers. To achieve this, the platforms compile the Wasm programs to native code, which is then executed in a sandboxed environment. Then, the host processes are capable of instantiating a new Wasm sandbox for each client function, executing it in response to individual user requests in a matter of nanoseconds. Utilizing WebAssembly enables these platforms to inherently isolate the execution of client functions from one another as well as from the host process. However, this isolation is not foolproof against Spectre attacks [? ?].

In the subsequent sections, we illustrate how diversification can be used to protect WebAssembly binaries against Spectre attacks. We show a case of Defensive Software Diversification for the sake of protecting WebAssembly binaries.

4.2.1 Threat model: speculative side-channel attacks

Let us illustrate the threat model in which a WebAssembly program could be susceptible in FaaS platforms. Developers can submit any WebAssembly binary to the FaaS platform. This includes potential malicious actors that could upload a Wasm binary that, when compiled to native code, uses Spectre attacks to leak sensitive information from the host process. Spectre attacks exploit hardware predictors to induce misspredictions and speculatively execute instructions—gadgets that would not run sequentially. The attacker could then use this information to infer the contents of the memory of other client functions, or even the host process itself.

Narayan and colleagues [?] dissected the possible Spectre attacks for Wasm binaries into three categories based on the specific hardware predictor that is exploited and the specific FaaS scenario: Sandbox breakout attacks, Sandbox poisoning attacks and Host poisoning attacks. The first one, exploits the branch target buffer by predicting the target of an indirect jump, thereby rerouting speculative control flow to an arbitrary target. The second one takes advantage of the pattern history table to anticipate the direction of a conditional branch during the ongoing evaluation of a condition. The third and last one exploits the return stack buffer that stores the locations of recently executed call instructions to predict the target of `ret` instructions. Each attack methodology relies on the extraction of memory bytes from another hosted Wasm binary that executes in the same host process.

TODO Add diagram **TODO** Explain here the threat model with the diagram

4.2.2 Approach

- Use of wasm-mutate

4.2.3 Results

- Diminshing of BER - Rockiki paper on portable side channel in browsers.

TODO TBD discuss deoptimization

4.2.4 Partial input/output validation

When WASM-MUTATE generates a variant, it can be executed to check the input/output equivalence. If the variant has a _start function, both binaries, the original and the variant can be initialized. If the state of the memory, the globals and the stack is the same after executing the _start function, they are partially equivalent.

The _start function is easier to execute given its signature. It does not receive parameters. Therefore, it can be executed directly. Yet, since a WebAssembly program might contain more than one function that could be indistinctly called with and arbitrary number of parameters, we are not able to validate the whole program. Thus, we call the checking of the initialization of a Wasm variant, a partial validation.

4.2.5 Some other works to be cited along with the paper. Mostly in the Intro

Spectre and side-channel defenses

- paper 2021: Read this, since it is super related, `https://www.isecure-journal.com/article_136367_a3948a522c7c59c65b65fa87571fde7b.pdf` [**?**]

- A dataset of Wasm programs: [**?**]

- Papers 2020

- Papers 2019 - [**?**]

Selwasm: A code protection mechanism for webassembly

Babble

- https://arxiv.org/pdf/2212.04596.pdf

Principled Composition of Function Variants for Dynamic Software Diversity and Program Protection

- https://dl.acm.org/doi/10.1145/3551349.3559553

How Far We've Come – A Characterization Study of Standalone WebAssembly Runtimes

- https://ieeexplore.ieee.org/document/9975423

Code obfuscation against symbolic execution attacks

Code artificiality: A metric for the code stealth based on an n-gram model

Semantics-aware obfuscation scheme prediction for binary

Wobfuscator: Obfuscating javascript malware via opportunistic translation to webassembly

Synthesizing Instruction Selection Rewrite Rules from RTL using SMT "We also synthesize integer rewrite rules from WebAssembly to RISC-V "

Wafl: Binary-only webassembly fuzzing with fast snapshots

Contribution paper

The case discussed in this section is fully detailed in Cabrera-Arteaga et al. "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly" *Under review* `https://arxiv.org/pdf/2309.07638.pdf`.

4.3 Conclusions

5 CONCLUSIONS AND FUTURE WORK

5.1 Summary of technical contributions

5.2 Summary of empirical findings

5.3 Future Work

Furthermore, WASM-MUTATE can benefit from the enumerative synthesis techniques employed by CROW and MEWE. Specifically, WASM-MUTATE could incorporate the transformations generated by these tools as rewriting rules.

Moreover, the WebAssembly ecosystem is still in its infancy compared to more mature programming environments. A 2021 study by Hilbig et al. found only 8,000 unique WebAssembly binaries globally[**?**], a fraction of the 1.5 million and 1.7 million packages available in npm and PyPI, respectively. This limited dataset poses challenges for machine learning-based analysis tools, which require extensive data for effective training. The scarcity of WebAssembly programs also exacerbates the problem of software monoculture, increasing the risk of compromised WebAssembly programs being consumed[**?**]. This dissertation aims to mitigate these issues by introducing a comprehensive suite of tools designed to enhance WebAssembly security through Software Diversification and to improve testing rigor within the ecosystem.

Program Normalization WASM-MUTATE was previously employed successfully for the evasion of malware detection, as outlined in [**?**]. The proposed mitigation in the prior study involved code normalization as a means of reducing the spectrum of malware variants. Our current work provides insights into the potential effectiveness of this approach. Specifically, a practically costless process of pre-compiling Wasm binaries could be employed as a preparatory measure for malware classifiers. In other words, a Wasm binary can first be compiled with

[0] Comp. time 2023/10/06 12:22:33

wasmtime, effectively eliminating approx. 25% of malware variants according to our preservation statistics for wasmtime. This approach could substantially enhance the efficiency and precision of malware detection systems.

REFERENCES

[1] A. Haas, A. Rossberg, D. L. Schuff, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien, “Bringing the web up to speed with webassembly,” *PLDI*, 2017.

[2] A. Rossberg, B. L. Titzer, A. Haas, D. L. Schuff, D. Gohman, L. Wagner, A. Zakai, J. F. Bastien, and M. Holman, “Bringing the web up to speed with webassembly,” *Commun. ACM*, vol. 61, p. 107–115, nov 2018.

[3] P. Mendki, “Evaluating webassembly enabled serverless approach for edge computing,” in *2020 IEEE Cloud Summit*, pp. 161–166, 2020.

[4] M. Jacobsson and J. Wåhslén, “Virtual machine execution for wearables based on webassembly,” in *EAI International Conference on Body Area Networks*, pp. 381–389, Springer, Cham, 2018.

[5] “Webassembly system interface.” `https://github.com/WebAssembly/WASI`, 2021.

[6] D. Bryant, “Webassembly outside the browser: A new foundation for pervasive computing,” in *Proc. of ICWE 2020*, pp. 9–12, 2020.

[7] B. Spies and M. Mock, “An evaluation of webassembly in non-web environments,” in *2021 XLVII Latin American Computing Conference (CLEI)*, pp. 1–10, 2021.

[8] E. Wen and G. Weber, “Wasmachine: Bring iot up to speed with a webassembly os,” in *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pp. 1–4, IEEE, 2020.

[9] D. Lehmann, J. Kinder, and M. Pradel, “Everything old is new again: Binary security of webassembly,” in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020.

[10] M. Kim, H. Jang, and Y. Shin, “Avengers, assemble! survey of webassembly security solutions,” in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pp. 543–553, 2022.

[11] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, “Sledge: A serverless-first, light-weight wasm runtime for the edge,” in *Proceedings of the 21st International Middleware Conference*, p. 265–279, 2020.

[12] R. Gurdeep Singh and C. Scholliers, “Warduino: A dynamic webassembly virtual machine for programming microcontrollers,” in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2019, (New York, NY, USA), pp. 27–36, ACM, 2019.

[13] S. S. Salim, A. Nisbet, and M. Luján, “Trufflewasm: A webassembly interpreter on graalvm,” in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE ’20, (New York, NY, USA), p. 88–100, Association for Computing Machinery, 2020.

[14] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown, “Wave: a verifiably secure webassembly sandboxing runtime,” in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2940–2955, 2023.

[15] Q. Stiévenart and C. De Roover, “Wassail: a webassembly static analysis library,” in *Fifth International Workshop on Programming Technology for the Future Web*, 2021.

[16] I. Bastys, M. Algehed, A. Sjösten, and A. Sabelfeld, “Secwasm: Information flow control for webassembly,” in *Static Analysis* (G. Singh and C. Urban, eds.), (Cham), pp. 74–103, Springer Nature Switzerland, 2022.

[17] T. Brito, P. Lopes, N. Santos, and J. F. Santos, “Wasmati: An efficient static vulnerability scanner for webassembly,” *Computers & Security*, vol. 118, p. 102745, 2022.

[18] F. Marques, J. Fragoso Santos, N. Santos, and P. Adão, “Concolic execution for webassembly (artifact),” Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[19] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, “, : Sfi safety for native-compiled wasm,” *Network and Distributed Systems Security (NDSS) Symposium.*

[20] W. Fu, R. Lin, and D. Inge, “Taintassembly: Taint-based information flow control tracking for webassembly,” *arXiv preprint arXiv:1802.01050*, 2018.

[21] D. Lehmann and M. Pradel, “Wasabi: A framework for dynamically analyzing webassembly,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1045–1058, 2019.

[22] D. Lehmann, M. T. Torp, and M. Pradel, “Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly,” *arXiv preprint arXiv:2110.15433*, 2021.

[23] Q. Stiévenart, D. Binkley, and C. De Roover, "Dynamic slicing of webassembly binaries," in *39th IEEE International Conference on Software Maintenance and Evolution*, IEEE, 2023.

[24] Q. Stiévenart, D. W. Binkley, and C. De Roover, "Static stack-preserving intra-procedural slicing of webassembly binaries," in *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, (New York, NY, USA), p. 2031–2042, Association for Computing Machinery, 2022.

[25] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "Ct-wasm: Type-driven secure cryptography for the web ecosystem," *Proc. ACM Program. Lang.*, vol. 3, jan 2019.

[26] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "New kid on the web: A study on the prevalence of webassembly in the wild," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*, pp. 23–42, Springer, 2019.

[27] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna, "Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1714–1730, 2018.

[28] A. Romano, Y. Zheng, and W. Wang, "Minerray: Semantics-aware analysis for ever-evolving cryptojacking detection," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1129–1140, 2020.

[29] F. N. Naseem, A. Aris, L. Babun, E. Tekiner, and A. S. Uluagac, "Minos: A lightweight real-time cryptojacking detection system.," in *NDSS*, 2021.

[30] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao, "Seismic: Secure in-lined script monitors for interrupting cryptojacks," in *Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II 23*, pp. 122–142, Springer, 2018.

[31] J. D. P. Rodriguez and J. Posegga, "Rapid: Resource and api-based detection against in-browser miners," in *Proceedings of the 34th Annual Computer Security Applications Conference*, pp. 313–326, 2018.

[32] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, N. Borisov, M. Antonakakis, and M. Bailey, "Outguard: Detecting in-browser covert cryptocurrency mining in the wild," in *The World Wide Web Conference*, pp. 840–852, 2019.

[33] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, “Drive-by key-extraction cache attacks from portable code,” *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 119, 2018.

[34] G. Maisuradze and C. Rossow, “Ret2spec: Speculative execution using return stack buffers,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’18, (New York, NY, USA), p. 2109–2122, Association for Computing Machinery, 2018.

[35] T. Rokicki, C. Maurice, M. Botvinnik, and Y. Oren, “Port contention goes portable: Port contention side channels in web browsers,” in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS ’22, (New York, NY, USA), p. 1182–1194, Association for Computing Machinery, 2022.

[36] Q. Stiévenart, C. De Roover, and M. Ghafari, “Security risks of porting c programs to webassembly,” in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC ’22, (New York, NY, USA), p. 1713–1722, Association for Computing Machinery, 2022.

[37] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, “N-variant systems: a secretless framework for security through diversity,” in *Proc. of USENIX Security Symposium*, USENIX-SS’06, 2006.

[38] J. Cabrera Arteaga, O. Floros, O. Vera Perez, B. Baudry, and M. Monperrus, “Crow: code diversification for webassembly,” in *MADWeb, NDSS 2021*, 2021.

[39] J. Cabrera Arteaga, “Artificial software diversification for webassembly,” 2022. QC 20220909.

[40] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. N. Saw, and R. Venkatesan, “The superdiversifier: Peephole individualization for software protection,” in *International Workshop on Security*, pp. 100–120, Springer, 2008.

[41] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, “Souper: A Synthesizing Superoptimizer,” *arXiv preprint 1711.04422*, 2017.

[42] J. Cabrera Arteaga, P. Laperdrix, M. Monperrus, and B. Baudry, “Multi-Variant Execution at the Edge,” *arXiv e-prints*, p. arXiv:2108.08125, Aug. 2021.

[43] S. Bhatkar, D. C. DuVarney, and R. Sekar, “Address obfuscation: an efficient approach to combat a board range of memory error exploits,” in *Proceedings of the USENIX Security Symposium*, 2003.

[44] J. Lettner, D. Song, T. Park, P. Larsen, S. Volckaert, and M. Franz, "Partisan: fast and flexible sanitization via run-time partitioning," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 403–422, Springer, 2018.

[45] B. G. Ryder, "Constructing the call graph of a program," *IEEE Transactions on Software Engineering*, no. 3, pp. 216–226, 1979.

[46] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, *et al.*, "Swivel: Hardening webassembly against spectre," in *USENIX Security Symposium*, 2021.

[47] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Sfi safety for native-compiled wasm," *NDSS. Internet Society*, 2021.

[48] J. Cabrera-Arteaga, N. Fitzgerald, M. Monperrus, and B. Baudry, "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly," *arXiv e-prints*, p. arXiv:2309.07638, Sept. 2023.

[49] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," *Proc. ACM Program. Lang.*, vol. 5, jan 2021.

[50] "Stop a wasm compiler bug before it becomes a problem | fastly." `https://www.fastly.com/blog/defense-in-depth-stopping-a-wasm-compiler-bug-before-it-became-a-problem`, 2021.

[51] D. Cao, R. Kunkel, C. Nandi, M. Willsey, Z. Tatlock, and N. Polikarpova, "Babble: Learning better abstractions with e-graphs and anti-unification," *Proc. ACM Program. Lang.*, vol. 7, jan 2023.

[52] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, (New York, NY, USA), p. 264–276, Association for Computing Machinery, 2009.

[53] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 1–11, IEEE, 2013.

[54] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, "Swivel: Hardening WebAssembly against spectre," in *30th USENIX Security*

Symposium (USENIX Security 21), pp. 1433–1450, USENIX Association, Aug. 2021.

[55] T. D. Morgan and J. W. Morgan, “Web timing attacks made practical,” *Black Hat*, 2015.

[56] T. Schnitzler, K. Kohls, E. Bitsikas, and C. Pöpper, “Hope of delivery: Extracting user locations from mobile instant messengers,” in *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*, The Internet Society, 2023.

[57] S. Cao, N. He, Y. Guo, and H. Wang, “WASMixer: Binary Obfuscation for WebAssembly,” *arXiv e-prints*, p. arXiv:2308.03123, Aug. 2023.

[58] A. Hilbig, D. Lehmann, and M. Pradel, “An empirical study of real-world webassembly binaries: Security, languages, use cases,” *Proceedings of the Web Conference 2021*, 2021.

[59] Kaspersky, “The state of cryptojacking in the first three quarters of 2022,” 2022.

[60] Mozilla, “Protections Against Fingerprinting and Cryptocurrency Mining Available in Firefox Nightly and Beta ,” 2019.

[61] S. Bhansali, A. Aris, A. Acar, H. Oz, and A. S. Uluagac, “A first look at code obfuscation for webassembly,” in *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec ’22, (New York, NY, USA), p. 140–145, Association for Computing Machinery, 2022.

[62] J. Cabrera-Arteaga, M. Monperrus, T. Toady, and B. Baudry, “Webassembly diversification for malware evasion,” *Computers & Security*, vol. 131, p. 103296, 2023.

[63] E. Tekiner, A. Acar, A. S. Uluagac, E. Kirda, and A. A. Selcuk, “In-browser cryptomining for good: An untold story,” in *2021 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, pp. 20–29, 2021.

[64] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1–19, 2019.

Part II

Included papers

SUPEROPTIMIZATION OF WEBASSEMBLY BYTECODE

Javier Cabrera-Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus

Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs

https://doi.org/10.1145/3397537.3397567

CROW: CODE DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus

Network and Distributed System Security Symposium (NDSS 2021), MADWeb

https://doi.org/10.14722/madweb.2021.23004

MULTI-VARIANT EXECUTION AT THE EDGE

Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry

Conference on Computer and Communications Security (CCS 2022), Moving Target Defense (MTD)

`https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

WEBASSEMBLY DIVERSIFICATION FOR MALWARE EVASION

Javier Cabrera-Arteaga, Tim Toady, Martin Monperrus, Benoit Baudry

Computers & Security, Volume 131, 2023

`https://www.sciencedirect.com/science/article/pii/S0167404823002067`

WASM-MUTATE: FAST AND EFFECTIVE BINARY DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Nick Fitzgerald, Martin Monperrus, Benoit Baudry

Under revision

SCALABLE COMPARISON OF JAVASCRIPT V8 BYTECODE TRACES

Javier Cabrera-Arteaga, Martin Monperrus, Benoit Baudry

11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019)

`https://doi.org/10.1145/3358504.3361228`